UPC-IO: A Parallel I/O API for UPC

V1.0

Tarek El-Ghazawi
François Cantonnet
Proshanta Saha
The George Washington University
801 22nd Street NW • Suite 607
Washington,DC 20052, USA
{tarek, fcantonn, sahap}@gwu.edu

Rajeev Thakur
Rob Ross
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439, USA
{thakur, rross}@mcs.anl.gov

Dan Bonachea
Dept. of Computer Science
University of California, Berkeley
Berkeley, CA 94720, USA
bonachea@cs.berkeley.edu

July 29, 2004

Contents

3 Terms, definitions and symbols

Some of the following definitions are repeated from the UPC Language Specifications [1] for self-containment and clarity of the functions defined here.

3.1 Collective

1. A constraint placed on some language operations which requires evaluation of such operations to be matched across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.[1]

3.2 Single-valued

1. an operand to a collective operation, which has the same value on every thread. The behavior of the operation is otherwise undefined.

3.3 List Based File Access

1. File accesses done using explicit offsets and sizes of data. Non-contiguous accesses may be performed using lists of explicit offsets and lengths in the file.

3.4 File Pointer Based Access

1. File accesses are done using either individual or common file pointers, where individual file pointers provide a means for each thread to independently control its access to the file and common file pointers provide a means for all the threads to access the file with a single, collective file pointer.

[1]A collective operation need not provide any actual synchronization between threads, unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely upon such unspecified synchronization for correctness.

3.5 Synchronous I/O Call

1. I/O calls which block and wait until the corresponding I/O operation is completed.

3.6 Asynchronous I/O Call

1. I/O calls which start an I/O operation and return immediately, and must later be completed using a synchronization function. Only one outstanding asynchronous operation is allowed on a UPC-IO file handle at any given time.

3.7 Consistency Semantics

1. Consistency semantics define when the data written to a file by a thread is visible to other threads. The consistency semantics also define the outcome in the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions.

3.8 Atomicity Semantics

1. Atomicity semantics define the outcome of operations in which multiple threads write concurrently to a file and some of the writes overlap each other.

7 Library

7.3 UPC Parallel I/O <upc_io.h>

1. This subsection provides the UPC parallel extensions of Section 7.19 in [2]. All the characteristics of library functions described in section 7.1.4 in [2] apply to these as well.

Common Constraints

1. All UPC-IO functions are collective and must be called by all threads collectively. (See Section 3 of the UPC Specification [1] for the definition of collective).[2]

2. If a program calls `exit`, `upc_global_exit`, or returns from `main` with a UPC file still open, the file will automatically be closed at program termination, and the effect will be equivalent to `upc_all_fclose` being implicitly called on the file.

3. If a program attempts to read past the end of a file, the read function will read data up to the end of file and return the number of bytes actually read, which may be less than the amount requested.

4. Writing past the end of a file increases the file size.

5. If a program seeks to a location past the end of a file and writes starting from that location, the data in the intermediate (unwritten) portion of the file is undefined. For example, if a program opens a new file (of size 0 bytes), seeks to offset 1024 and writes some data beginning from that offset, the data at offsets 0–1023 is undefined. Seeking past the end of file and performing a write causes the current file size to immediately be extended up to the end of the write. However, just seeking past the end of file or attempting to read past the end of file, without a write, does not extend the file size.

6. All "`shared void *`" pointers passed to the I/O functions (as function arguments or indirectly through the list I/O arguments) are treated as if they had a phase field of zero (that is, the input phase is ignored).

[2]Note that collective does not necessarily imply barrier synchronization. The synchronization behavior of the UPC-IO data movement library functions is explicitly controlled by using the `sync_mode` flag argument. See Section 7.3.1.3 for details.

7. All UPC-IO read/write functions take an argument `sync_mode` of type `upc_flag_t`. `sync_mode` values are obtained by performing a bitwise `OR` of a constant of the form `UPC_IN_XSYNC` and a constant of the form `UPC_OUT_YSYNC`, where X and Y may be `NO`, `MY`, or `ALL`. The `sync_mode` argument is similar to the `sync_mode` argument used in collective operation functions[4]. The `sync_mode` argument and `upc_flag_t` type are further discussed in Section 7.3.1.3.

8. The arguments to all UPC-IO functions are single-valued (must have the same value on all threads) except where explicitly noted otherwise in the function description (See Section 3 of the UPC Specification [1] for the definition of single-valued).

9. UPC-IO, by default, supports weak consistency and atomicity semantics. The default (weak) semantics are as follows. The data written to a file by a thread is only guaranteed to be visible to another thread after all threads have collectively closed or synchronized the file.

10. Writes to a file from a given thread are always guaranteed to be visible to subsequent file reads by the *same* thread, even without an intervening call to collectively close or synchronize the file.

11. Byte-level data consistency is supported.

12. If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined with the weak consistency and atomicity semantics

13. When reading data being concurrently written by another thread, the data that gets read is undefined with the weak consistency and atomicity semantics.

14. File reads into overlapping locations in a shared buffer in memory using individual file pointers or list I/O functions leads to undefined data in the target buffer under the weak consistency and atomicity semantics.

15. A given file must not be opened at same time by the POSIX I/O and UPC-IO libraries.

16. Except where otherwise noted, all UPC-IO functions return NON-single-valued errors; that is, the occurrence of an error need only be

reported to at least one thread, and the `errno` value reported to each such thread may differ. When an error is reported to ANY thread, the position of ALL file pointers for the relevant file handle becomes undefined.

17. The error values that UPC-IO functions may set in `errno` are implementation-defined, however the `perror()` and `strerror()` functions are still guaranteed to work properly with `errno` values generated by UPC-IO.

18. UPC-IO functions can not be called between `upc_notify` and corresponding `upc_wait` statements.

7.3.0 Background

7.3.0.1 File Accessing and File Pointers

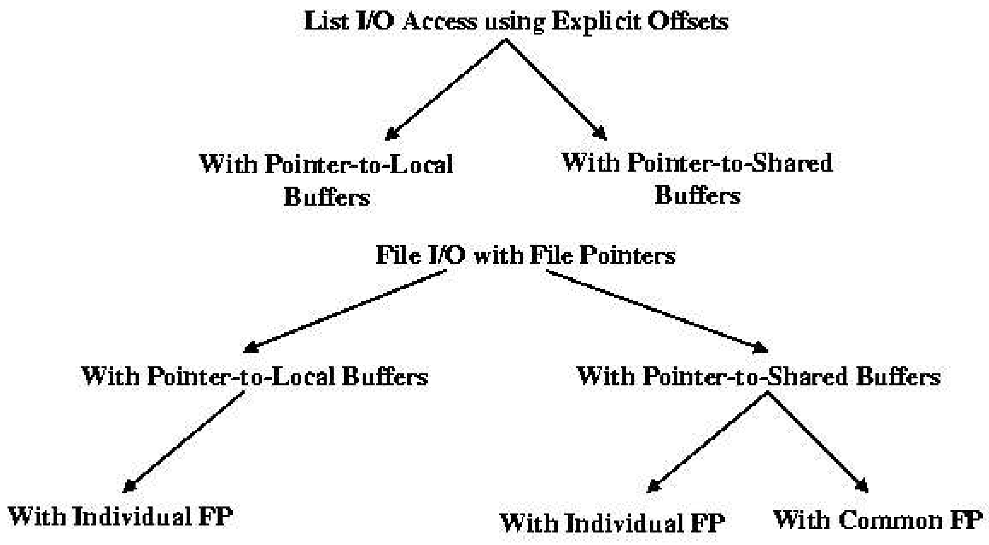
List I/O Access using Explicit Offsets
With Pointer-to-Local Buffers
With Pointer-to-Shared Buffers
File I/O with File Pointers
With Pointer-to-Local Buffers
With Pointer-to-Shared Buffers
With Individual FP
With Individual FP
With Common FP

Figure 1: UPC-IO File Access Methods

Collective UPC-IO accesses can be done in and out of shared and private buffers, thus local and shared reads and writes are generally supported. In each of these cases, file pointers could be either common or individual. Note that in UPC-IO, common file pointers cannot be used in conjunction with pointer-to-local buffers. File pointer modes are specified by passing a flag to the collective `upc_all_fopen` function and can be changed using `upc_all_fcntl`. When a file is opened with the common file pointer flag, all threads share a common file pointer. When a file is opened with the individual file pointer flag, each thread gets its own file pointer.

UPC-IO also provides file-pointer-independent list file accesses by specifying explicit offsets and sizes of data that is to be accessed. List IO can also be used with either pointer-to-local buffers or pointer-to-shared buffers.

Examples 1-3 and their associated figures, Figures 2-4, give typical instances of UPC-IO usage. Error checking is omitted for brevity.

```
Example 1:

double buffer[10];  // and assuming a total of 4 THREADS
upc_file_t *fd;

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL );
upc_all_fseek( fd, 5*MYTHREAD*sizeof(double), UPC_SEEK_SET );
upc_all_fread_local( fd, buffer, sizeof(double), 10,
                          UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
upc_all_fclose(fd);
```

Example 1 and Figure 2 illustrate a collective read operation using individual file pointers. Each thread reads a block of data into a private buffer from a particular thread-specific offset.

Example 2 and Figure 3 illustrate a collective read operation using a common file pointer. The data read is stored into a shared buffer, having a block size of 5 elements. The user selects the type of file pointer at file-open time. The user can select either individual file pointers by passing the flag `UPC_INDIVIDUAL_FP` to the function `upc_all_fopen`, or the common file pointer by passing the flag `UPC_COMMON_FP` to `upc_all_fopen`.

```
Example 2:

shared [5] float buffer[20];  // and assuming a total of 4 static THREADS
upc_file_t *fd;
```

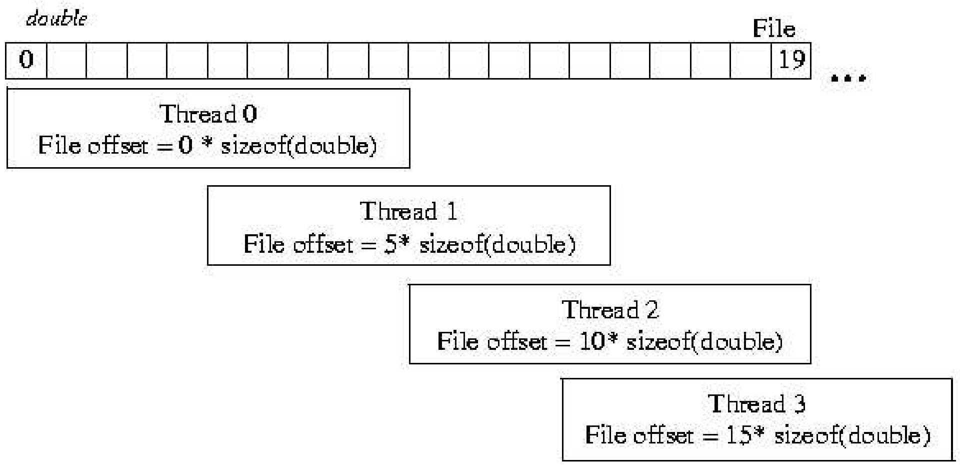
double
File
0
19
...
Thread 0
File offset = 0 * sizeof(double)
Thread 1
File offset = 5* sizeof(double)
Thread 2
File offset = 10* sizeof(double)
Thread 3
File offset = 15* sizeof(double)

Figure 2: Collective read into private buffers can provide a canonical file-view

```
fd = upc_all_fopen( "file", UPC_RDONLY | UPC_COMMON_FP, 0, NULL );
upc_all_fread_shared( fd, buffer, upc_blocksizeof(buffer),
     upc_elemsizeof(buffer), 20, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
/* or equivalently:
 *  upc_all_fread_shared( fd, buffer, 5, sizeof(float), 20,
                                 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
 */
```

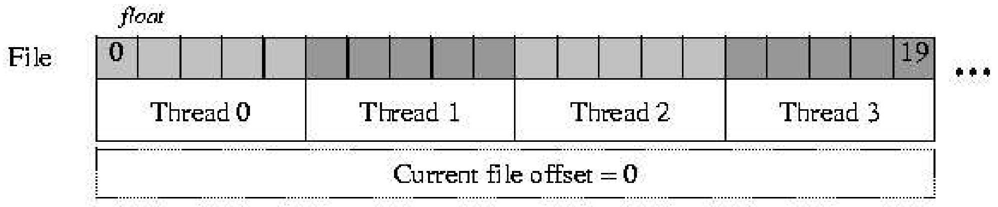

Figure 3: Collective read into a blocked shared buffer can provide a partitioned file-view

Example 3 and Figure 4 illustrate a collective list I/O read operation. The list I/O functions allow the user to specify noncontiguous accesses both in memory and file in the form of lists of explicit offsets and lengths in the file and explicit address and lengths in memory. None of the file pointers are used or updated in this case.

```
Example 3:

upc_local_memvec_t memvec[2];
upc_filevec_t filevec[2];
upc_file_t *fd;
char buffer[12];

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL );
memvec[0].baseaddr = &buffer[0];
memvec[0].len = 4;
memvec[1].baseaddr = &buffer[7];
memvec[1].len = 3;
filevec[0].offset = MYTHREAD*5;
filevec[0].len = 2;
filevec[1].offset = 10+MYTHREAD*5;
filevec[1].len = 5;

upc_all_fread_list_local( fd, 2, &memvec, 2, &filevec,
                                    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

7.3.0.2 Synchronous and Asynchronous I/O

I/O operations can be synchronous (blocking) or asynchronous (non-blocking). While synchronous calls are quite simple and easy to use from a programming point of view, asynchronous operations allow the overlapping of computation and I/O to achieve improved performance. Synchronous calls block and wait until the corresponding I/O operation is completed. On the other hand, an asynchronous call starts an I/O operation and returns immediately. Thus, the executing process can turn its attention to other processing needs while the I/O is progressing.

UPC-IO supports both synchronous and asynchronous I/O functionality. The asynchronous I/O functions have the same syntax and basic semantics as their synchronous counterparts, with the addition of the “async” suffix in

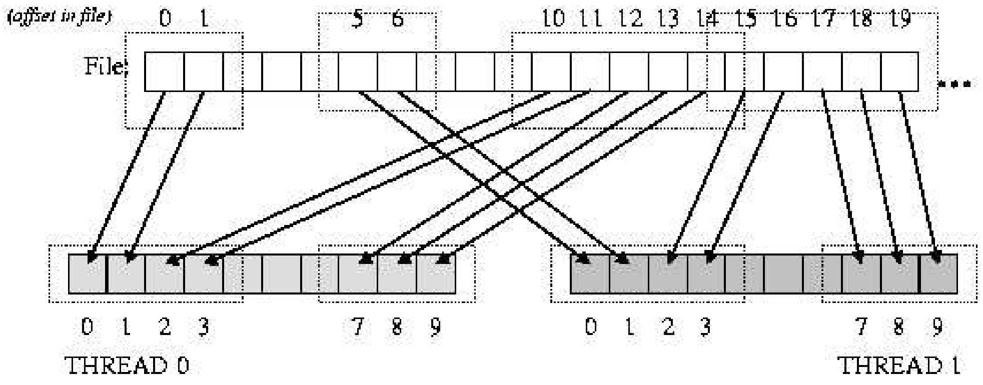

Figure 4: List I/O read of noncontiguous parts of a file to private noncontiguous buffers

their names. The asynchronous I/O functions have the restriction that only one (collective) asynchronous operation can be active at a time on a given file handle. That is, an asynchronous I/O function must be completed by calling `upc_all_ftest_async` or `upc_all_fwait_async` before another asynchronous I/O function can be called on the same file handle. This restriction is similar to the restriction MPI-IO [3] has on split-collective I/O functions: only one split collective operation can be outstanding on an MPI-IO file handle at any time.

7.3.0.3 Consistency and Atomicity Semantics

Let us first define what we mean by the terms consistency semantics and atomicity semantics. The consistency semantics define when the data written to a file by a thread is visible to other threads. The atomicity semantics define the outcome of operations in which multiple threads write concurrently to a file or shared buffer and some of the writes overlap each other. For performance reasons, UPC-IO uses weak consistency and atomicity semantics by default. The user can select stronger semantics either by opening the file with the flag `UPC_STRONG_CA` or by calling `upc_all_fcntl` with the command `UPC_SET_STRONG_CA_SEMANTICS`.

The default (weak) semantics are as follows. The data written by a thread is only guaranteed to be visible to another thread after all threads have called `upc_all_fclose` or `upc_all_fsync`. (Note that the data *may* be visible to other threads before the call to `upc_all_fclose` or `upc_all_fsync` and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_fclose` or `upc_all_fsync`. Byte-level data consistency is supported. So for example, if thread 0 writes one byte at offset 0 in the file and thread 1 writes one byte at offset 1 in the file, the data from both threads will get written to the file. If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined. Similarly, if one thread tries to read the data being concurrently written by another thread, the data that gets read is undefined. Concurrent in this context means any two read/write operations to the same file handle with no intervening calls to `upc_all_fsync` or `upc_all_fclose`.

For the functions that read into or write from a shared buffer using a common file pointer, the weak consistency semantics are defined as follows. Each call to `upc_all_{fread,fwrite}_shared[_async]` with a common file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different for each operation). In other words, NO file reads are guaranteed to see the results of file writes using the common file pointer until after a close or sync under the default weak consistency semantics.

By passing the `UPC_STRONG_CA` flag to `upc_all_fopen` or by calling `upc_all_fcntl` with the command `UPC_SET_STRONG_CA_SEMANTICS`, the user selects strong consistency and atomicity semantics. In this case, the data written by a thread is visible to other threads as soon as the file write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. Overlapping writes to a file in a single (list I/O) write function on a single thread are not permitted (see Section 7.3.5). While strong consistency and atomicity semantics are selected on a given file handle, the `sync_mode` argument to all fread/fwrite functions on that handle is ignored and always treated as `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`.

The consistency semantics also define the outcome in the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions. By default, the data in the overlapping space is undefined. If the user selects strong consistency and atomicity, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order. Overlapping reads into memory buffers in a single (list I/O) read function on a single thread are not permitted (see Section 7.3.5).

Note that in strong consistency and atomicity mode, atomicity is guaranteed at the UPC-IO function level. The entire operation specified by a single function is performed atomically, regardless of whether it represents a single, contiguous read/write or multiple noncontiguous reads or writes as in a list I/O function.

Example

Consider the following example in which three threads write data to a file concurrently, each with a single list I/O function. The numbers indicate file offsets and brackets indicate the boundaries of a listed vector. Each thread writes its own thread id as the data values:

```
thread 0:    {1  2  3}   {5  6  7  8}
thread 1: {0  1  2}{3  4  5}
thread 2:             {4  5  6}   {8  9 10 11}
```

With the default weak semantics, the results in the overlapping locations are undefined. Therefore, the result in the file would be the following, where x represents undefined data.

```
File:     1  x   x  x  x  x  x  0  x  2  2  2
```

That is, the data from thread 1 is written at location 0, the data from thread 0 is written at location 7, and the data from thread 2 is written at locations 9, 10, and 11, because none of these locations had overlapping writes. All other locations had overlapping writes, and consequently, the result at those locations is undefined.

If the file were opened with the `UPC_STRONG_CA` flag, strong consistency and atomicity semantics would be in effect. The result, then, would depend on the order in which the writes from the three threads actually occurred. Since six different orderings are possible, there can be six outcomes. Let us assume, for example, that the ordering was the write from thread 0, followed by the write from thread 2, and then the write from thread 1. The (list I/O) write from each thread happens atomically. Therefore, for this ordering, the result would be:

```
File:     1  1  1  1  1  1  2  0  2  2  2  2
```

We note that if instead of using a single list I/O function each thread used a separate function to write each contiguous portion, there would be six write functions, two from each thread, and the atomicity would be at the granularity of the write operation specified by each of those functions.

7.3.0.4 File Interoperability

UPC-IO does not specify how an implementation may store the data in a file on the storage device. Accordingly, it is implementation-defined whether or not a file created by UPC-IO can be directly accessed by using C/POSIX I/O functions. However, the UPC-IO implementation must specify how the user can retrieve the file from the storage system as a linear sequence of bytes and vice versa. Similarly, the implementation must specify how familiar operations, such as the equivalent of POSIX `ls`, `cp`, `rm`, and `mv` can be performed on the file.

7.3.1 Predefined Types

7.3.1.1 The `upc_off_t` type

Synopsis

1.
```
#include <upc_io.h>

upc_off_t myOffset;
```

Description

1. `upc_off_t` is a signed integral type that is capable of representing the size of the largest file supported by the implementation.

7.3.1.2 The `upc_file_t` type

Synopsis

1.
```
#include <upc_io.h>

upc_file_t *myFile;
```

Description

1. An opaque shared data type of incomplete type (as defined in section 6.2.5 of [2]) that represents an open file handle:
   `upc_file_t`

Constraints

1. `upc_file_t` objects are always manipulated via a pointer (that is, `upc_file_t *`).
2. `upc_file_t` is a shared data type. It is legal to pass a (`upc_file_t *`) across threads, and two pointers to `upc_file_t` that reference the same logical file handle will always compare equal.

Advice to implementors

1. The definition of `upc_file_t` does not restrict the implementation to store all its metadata with affinity to one thread. Each thread can still have local access to its metadata. For example, below is a simple approach an implementation could use:

```
/* for a POSIX-based implementation */
typedef int my_internal_filehandle_t;

#ifdef _UPC_INTERNAL
  typedef struct _local_upc_file_t  {
    my_internal_filehandle_t fd;
    ... other metadata ...
  } local_upc_file_t;
#else
  struct _local_upc_file_t;
#endif

typedef shared struct _local_upc_file_t upc_file_t;

upc_file_t *upc_all_fopen(...) {

   upc_file_t *handles =
            upc_all_alloc(THREADS, sizeof(upc_file_t));

   /* get my handle */
   upc_file_t *myhandle = &(handles[MYTHREAD]);

   /* cast to a pointer-to-local */
   local_upc_file_t* mylocalhandle = (local_upc_file_t*)myhandle;

   /* setup my metadata using pointer-to-local */
   mylocalhandle->fd = open(...);

   ...

   return handles;
}
```

The basic idea is that the "handle" exposed to the user actually points to a cyclic, distributed array. As a result, each thread has easy, local access to its own internal handle metadata with no communication, while maintaining the property that the handle that UPC-IO exposes to the client is a single-valued pointer-to-shared. An additional advantage is that a thread can directly access the metadata for other threads, which may occasionally be desirable in the implementation.

7.3.1.3 The `upc_flag_t` type

Synopsis

1. `#include <upc_io.h>`

   `upc_flag_t sync_mode;`

Description

The `sync_mode` argument is similar to the corresponding argument in collective operation functions [4].

1. If the `sync_mode` has the value `(UPC_IN_XSYNC | UPC_OUT_YSYNC)`, then if `X` is

`NO` the function may begin to read or write data when the first thread has entered the I/O function call,

`MY` the function may begin to read or write only data which has affinity to threads that have entered the collective function call, and

`ALL` the function may begin to read or write data only after all threads have entered the collective function call[3]

and if `Y` is

`NO` the function may read and write data until the last thread has returned from the collective function call,

`MY` the function call may return in a thread only after all reads and writes of data with affinity to the thread are complete,[4] and

`ALL` the function call may return only after all reads and writes of data are complete.[5]

2. `UPC_IN_XSYNC` alone is equivalent to (`UPC_IN_XSYNC | UPC_OUT_ALLSYNC`), where `X` is `NO, MY,` or `ALL`.

3. `UPC_OUT_XSYNC` alone is equivalent to (`UPC_IN_ALLSYNC | UPC_OUT_XSYNC`), where `X` is `NO, MY,` or `ALL`.

4. 0 is equivalent to (`UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`).

5. In the `sync_mode` definitions above, "data" refers exclusively to data residing in user-owned memory buffers passed as arguments to the library function. In other words, the `sync_mode` flag only governs the behavior of library accesses to memory locations in user-provided buffers — it

[3]`UPC_IN_ALLSYNC` requires the collective I/O function to guarantee that after all threads have entered the collective function call all threads will read the same values of the input data.

[4]`UPC_OUT_MYSYNC` requires the collective I/O function to guarantee that after a thread returns from the collective function call the thread will not read any earlier values of the output data with affinity to that thread.

[5]`UPC_OUT_ALLSYNC` requires the collective I/O function to guarantee that after a thread returns from the collective function call the thread will not read any earlier values of the output data. `UPC_OUT_ALLSYNC` is not required to provide an "implied" barrier. For example, if the entire collective I/O operation has been completed by a certain thread before some other threads have reached their corresponding function calls, then that thread may exit its call.

does not restrict the behavior of read/write operations on the storage medium or any buffer memory internal to the library implementation.

6. The semantics of these flags when applied to the async variants of the fread/fwrite functions should be interpreted as follows: constraints that reference entry to a function call correspond to entering the fread_async-/fwrite_async call that initiates the asynchronous operation, and constraints that reference returning from a function call correspond to returning from the `upc_all_fwait_async()` or successful `upc_all_ftest_async()` call that completes the asynchronous operation. Also, note that all the `sync_mode` flags which govern an asynchronous operation are passed to the library during the asynchronous initiation call.

7. The `sync_mode` flag is included even on the fread/fwrite_local functions (which take a pointer-to-local as the buffer argument) in order to provide well-defined semantics for the case where one or more of the pointer-to-local arguments references a shared object (with local affinity). In the case where all of the pointer-to-local arguments in a given call reference only private objects, the `sync_mode` flag provides no useful additional guarantees and is recommended to be passed as `UPC_IN_NOSYNC|UPC_OUT_NOSYNC` to maxi mize performance.

7.3.1.4 The `upc_local_memvec_t` type

Synopsis

1.
```
#include <upc_io.h>


upc_local_memvec_t myLocalMemoryVector;
```

Description

1. `upc_local_memvec_t` is defined as follows:

```
typedef struct {
    void *baseaddr;
    size_t len;
} upc_local_memvec_t;
```

`baseaddr` and `len` specify a contiguous memory region in terms of the base address and length in bytes. `len` may be zero, in which case that entry is ignored.

7.3.1.5 The `upc_shared_memvec_t` type

Synopsis

1.
```
#include <upc_io.h>

upc_shared_memvec_t mySharedMemoryVector;
```

Description

1. `upc_shared_memvec_t` is defined as follows:

```
typedef struct {
    shared void *baseaddr;
    size_t blocksize;
    size_t len;
} upc_shared_memvec_t;
```

   `baseaddr` and `len` specify a shared memory region in terms of the base address and length in bytes. `len` may be zero, in which case that entry is ignored. `blocksize` is the block size of the shared buffer in bytes. A `blocksize` of 0 indicates an indefinite blocking factor.

7.3.1.6 The `upc_filevec_t` type

Synopsis

1.
```
#include <upc_io.h>

upc_filevec_t myFileVector;
```

Description

1. `upc_filevec_t` is defined as follows:

```
typedef struct {
    upc_off_t offset;
    size_t len;
} upc_filevec_t;
```

`offset` and `len` specify a contiguous region in the file in terms of the starting offset in the file in bytes and the length in bytes.

7.3.1.7 The `upc_hint_t` type

Synopsis

1. `#include <upc_io.h>`

   `upc_hint_t myHint;`

Description

1. `upc_hint_t` is defined as follows:

   ```
   typedef struct {
       const char *key;
       const char *value;
   } upc_hint_t;
   ```

2. UPC-IO supports a number of predefined hints. An implementation is free to support additional hints. An implementation is free to ignore any hint provided by the user. Implementations should *silently* ignore any hints they do not support or recognize. The predefined hints and their meanings are defined below. An implementation is not required to interpret these hint keys, but if it does interpret the hint key, it must provide the functionality described. For each hint name introduced, we describe the type of the hint value and its meaning. All hints are single-valued character strings (the content is single-valued, not the location).

`access_style` (comma-separated list of strings): indicates the manner in which the file is expected to be accessed. The hint value is a comma-separated list of any the following: "read_once", "write_once", "read_mostly", "write_mostly", "sequential", and "random". Passing such a hint does not place any constraints on how the file may actually be accessed by the program, although accessing the file in a way that is different from the specified hint may result in lower performance.

`collective_buffering` (boolean): specifies whether the application may benefit from collective buffering optimizations. Legal values for this key are "true" and "false". Collective buffering parameters can be further directed via additional hints: cb_buffer_size, and cb_nodes.

`cb_buffer_size` (decimal integer): specifies the total buffer space that the implementation can use on each thread for collective buffering.

`cb_nodes` (decimal integer): specifies the number of target threads or I/O nodes to be used for collective buffering.

`file_perm` (string): specifies the file permissions to use for file creation. The set of legal values for this key is implementation defined.

`io_node_list` (comma separated list of strings): specifies the list of I/O devices that should be used to store the file and is only relevant when the file is created.

`nb_proc` (decimal integer): specifies the number of threads that will typically be used to run programs that access this file and is only relevant when the file is created.

`striping_factor` (decimal integer): specifies the number of I/O devices that the file should be striped across and is relevant only when the file is created.

`start_io_device` (decimal integer): specifies the number of the first

I/O device from which to start striping the file and is relevant only when the file is created.

`striping_unit` (decimal integer): specifies the striping unit to be used for the file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

7.3.2 UPC File Operations

Common Constraints

1. When a file is opened with an individual file pointer, each thread will get its own file pointer and advance through the file at its own pace.
2. When a common file pointer is used, all threads positioned in the file will be aligned with that common file pointer.
3. Common file pointers cannot be used in conjunction with pointers-to-local (and hence cannot operate on private objects).
4. No function in this section (7.3.2) may be called while an asynchronous operation is pending on the file handle, except where otherwise noted.

7.3.2.1 The `upc_all_fopen` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

upc_file_t *upc_all_fopen(const char *fname,
                          int flags,
                          size_t numhints,
                          upc_hint_t const *hints)
```

Description

1. `upc_all_fopen` opens the file identified by the filename `fname` for input/output operations.

2. The flags parameter specifies the access mode. The valid flags and their meanings are listed below. Of these flags, exactly one of `UPC_RDONLY`, `UPC_WRONLY`, or `UPC_RDWR`, and one of `UPC_COMMON_FP` or `UPC_INDIVIDUAL_FP`, must be used. Other flags are optional. Multiple flags can be combined by using the bitwise OR operator (|), and each flag has a unique bitwise representation that can be unambiguously tested using the bitwise AND operator(&).

`UPC_RDONLY`	Open the file in read-only mode
`UPC_WRONLY`	Open the file in write-only mode
`UPC_RDWR`	Open the file in read/write mode
`UPC_INDIVIDUAL_FP`	Use an individual file pointer for all file accesses (other than list I/O)
`UPC_COMMON_FP`	Use the common file pointer for all file accesses (other than list I/O)
`UPC_APPEND`	Set the *initial* position of the file pointer to end of file. (The file pointer is not moved to the end of file after each read/write.)
`UPC_CREATE`	Create the file if it does not already exist
`UPC_EXCL`	Used in conjunction with `UPC_CREATE`. The open will fail if the file already exists.
`UPC_STRONG_CA`	Set strong consistency and atomicity semantics
`UPC_TRUNC`	Open the file and truncate it to zero length

`UPC_DELETE_ON_CLOSE` Delete the file automatically on close

3. The `UPC_COMMON_FP` flag specifies that all accesses (except for the list I/O operations) will use the common file pointer. The `UPC_INDIVIDUAL_FP` flag specifies that all accesses will use individual file pointers (except for the list I/O operations). Either `UPC_COMMON_FP` or `UPC_INDIVIDUAL_FP` must be specified or `upc_all_fopen` will return an error.

4. The `UPC_STRONG_CA` flag specifies strong consistency and atomicity semantics. The data written by a thread is visible to other threads as soon as the write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. In the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order.

5. If the flag `UPC_STRONG_CA` is not specified, weak semantics are provided. The data written by a thread is only guaranteed to be visible to another thread after all threads have called `upc_all_fclose` or `upc_all_fsync`. (Note that the data *may* be visible to other threads before the call to `upc_all_fclose` or `upc_all_fsync` and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_fclose` or `upc_all_fsync`. Byte-level data consistency is supported. For the purposes of atomicity and consistency semantics, each call to `upc_all_{fread,fwrite}_shared[_async]` with a common file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different for each operation)."[6]

6. Hints can be passed to the UPC-IO library as an array of key-value pairs[7] of strings. `numhints` specifies the number of hints in the `hints`

[6]In other words, NO reads are guaranteed to see the results of writes using the common file pointer until after a close or sync when `UPC_STRONG_CA` is not specified.

[7]The contents of the key/value pairs passed by all the threads must be single-valued.

array; if `numhints` is zero, the `hints` pointer is ignored. The user can free the `hints` array and associated character strings as soon as the open call returns. Each element of the `hints` array is of type `upc_hint_t`.

7. A file on the storage device is in the *open* state from the beginning of a successful open call to the end of the matching successful close call on the file handle. It is erroneous to have the same file *open* simultaneously with two `upc_all_fopen` calls, or with a `upc_all_fopen` call and a POSIX/C `open` or `fopen` call.

8. The user is responsible for ensuring that the file referenced by the `fname` argument refers to a single UPC-IO file. The actual argument passed on each thread may be different because the file name spaces may be different on different threads, but they must all refer to the same logical UPC-IO file.

9. On success, the function returns a pointer to a file handle that can be used to perform other operations on the file.

10. `upc_all_fopen` provides single-valued errors - if an error occurs, the function returns `NULL` on ALL threads, and sets `errno` appropriately to the same value on all threads.

7.3.2.2 The `upc_all_fclose` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

int upc_all_fclose(upc_file_t *fd);
```

Description

1. `upc_all_fclose` executes an implicit `upc_all_fsync` on `fd` and then closes the file associated with `fd`.

2. The function returns 0 on success. If `fd` is not valid or if an outstanding asynchronous operation on `fd` has not been completed, the function will return an error.

3. `upc_all_fclose` provides single-valued errors - if an error occurs, the function returns –1 on ALL threads, and sets `errno` appropriately to the same value on all threads.

4. After a file has been closed with `upc_all_fclose`, the file can legally be opened and the data in it can be accessed by using regular C/POSIX I/O calls.

7.3.2.3 The `upc_all_fsync` function

Synopsis

1.
   ```
   #include <upc.h>
   #include <upc_io.h>

   int upc_all_fsync(upc_file_t *fd)
   ```

Description

1. `upc_all_fsync` ensures that any data that has been written to the file associated with `fd` but not yet transferred to the storage device is transferred to the storage device. It also ensures that subsequent file reads from any thread will see any previously written values (that have not yet been overwritten).

2. There is an implied barrier immediately before `upc_all_fsync` returns.

3. The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.4 The `upc_all_fseek` function

Synopsis

1.
   ```
   #include <upc.h>
   #include <upc_io.h>
   ```

```
upc_off_t upc_all_fseek(upc_file_t *fd,
                        upc_off_t offset,
                        int origin)
```

Description

1. `upc_all_fseek` sets the current position of the file pointer associated with `fd`.
2. This offset can be relative to the current position of the file pointer, to the beginning of the file, or to the end of the file. The offset can be negative, which allows seeking backwards.
3. The origin parameter can be specified as `UPC_SEEK_SET`, `UPC_SEEK_CUR`, or `UPC_SEEK_END`, respectively, to indicate that the offset must be computed from the beginning of the file, the current location of the file pointer, or the end of the file.
4. In the case of a common file pointer, all threads must specify the same offset and origin. In the case of an individual file pointer, each thread may specify a different offset and origin.
5. It is legal to seek past the end of file. It is erroneous to seek to a negative position in the file. See the Common Constraints number 5 at the beginning of Section 7.3 for more details.
6. The current position of the file pointer can be determined by calling `upc_all_fseek(fd, 0, UPC_SEEK_CUR)`.
7. On success, the function returns the current location of the file pointer in bytes. If there is an error, it returns –1 and sets `errno` appropriately.

7.3.2.5 The `upc_all_fset_size` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

int upc_all_fset_size(upc_file_t *fd,
                      upc_off_t size)
```

Description

1. `upc_all_fset_size` executes an implicit `upc_all_fsync` on `fd` and resizes the file associated with `fd`.

2. `size` is measured in bytes from the beginning of the file.

3. If `size` is less than the current file size, the file is truncated at the position defined by `size`. The implementation is free to deallocate file blocks located beyond this position.

4. If `size` is greater than the current file size, the file size increases to `size`. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (between the old size and size) are undefined.

5. If this function truncates a file, it is possible that the individual and common file pointers may point beyond the end of file. This is legal and is equivalent to seeking past the end of file (see the Common Rules in Section 5 for the semantics of seeking past the end of file).

6. It is unspecified whether and under what conditions this function actually allocates file space on the storage device. Use `upc_all_fpreallocate` to force file space to be reserved on the storage device.

7. Calling this function does not affect the individual or common file pointers.

8. The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.6 The `upc_all_fget_size` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

upc_off_t upc_all_fget_size(upc_file_t *fd)
```

Description

1. `upc_all_fget_size` returns the current size in bytes of the file associated with `fd` on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.7 The `upc_all_fpreallocate` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

int upc_all_fpreallocate(upc_file_t *fd,
                         upc_off_t size)
```

Description

1. `upc_all_fpreallocate` ensures that storage space is allocated for the first `size` bytes of the file associated with `fd`.

2. Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, `upc_all_fpreallocate` has the same effect as writing undefined data.

3. If `size` is greater than the current file size, the file size increases to `size`. If `size` is less than or equal to the current file size, the file size is unchanged.

4. Calling this function does not affect the individual or common file pointers.

5. The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.8 The `upc_all_fcntl` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>
```

```
int upc_all_fcntl(upc_file_t *fd,
                  int cmd,
                  void *arg)
```

Description

1. `upc_all_fcntl` performs one of various miscellaneous operations related to the file specified by `fd`, as determined by `cmd`. The valid commands `cmd` and their associated argument `arg` are explained below.

`UPC_GET_CA_SEMANTICS`	Get the current consistency and atomicity semantics for `fd`. The argument `arg` is ignored. The return value is `UPC_STRONG_CA` for strong consistency and atomicity semantics and 0 for the default weak consistency and atomicity semantics.
`UPC_SET_WEAK_CA_SEMANTICS`	Executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the weak consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_SET_STRONG_CA_SEMANTICS`	Executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the strong consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_GET_FP`	Get the type of the current file pointer for `fd`. The argument `arg` is ignored. The return value is either `UPC_COMMON_FP` in case of a common file pointer, or `UPC_INDIVIDUAL_FP` for individual file pointers.
`UPC_SET_COMMON_FP`	Executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `fd` to a common file pointer (or leaves it unchanged if that mode is already selected), and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_SET_INDIVIDUAL_FP`	Executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `fd` to an individual file pointer (or leaves the mode unchanged if that mode is already selected), and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_GET_FL`	Get all the flags specified during the `upc_all_fopen` call for `fd`, as modified by any subsequent mode changes using the `upc_all_fcntl(UPC_SET_*)` commands. The argument `arg` is ignored. The return value has same format as the `flags` parameter in `upc_all_fopen`.
`UPC_GET_FN`	Get the file name provided by each thread in the `upc_all_fopen` call that created `fd`. The argument `arg` is a valid (`const char**`) pointing to a (`const char*`) location in which a pointer to file name will be written. Writes a (`const char*`) into `*arg` pointing to the filename in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fcntl` call on that file handle.

`UPC_GET_HINTS`	Retrieve the hints applicable to `fd`. The argument `arg` is a valid (`const upc_hint_t**`) pointing to a (`const upc_hint_t*`) location in which a pointer to the hints array will be written. Writes a (`const upc_hint_t*`) into `*arg` pointing to an array of `upc_hint_t`'s in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fnctl` call on that file handle. The number of hints in the array is returned by the call. The hints in the array may be a subset of those specified at file open time, if the implementation ignored some unrecognized or unsupported hints.
`UPC_SET_HINT`	Executes an implicit `upc_all_fsync` on `fd` and sets a new hint to `fd`. The argument `arg` points to one single-valued `upc_hint_t` hint to be applied. If the given hint key has already been applied to `fd`, the current value for that hint is replaced with the provided value. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_ASYNC _OUTSTANDING`	Returns 1 if there is an asynchronous operation outstanding on `fd`, or 0 otherwise.

2. In case of a non valid `fd`, `upc_all_fcntl` returns -1 and sets `errno` appropriately.

3. It *is* legal to call `upc_all_fcntl(UPC_ASYNC_OUTSTANDING)` when an asynchronous operation is outstanding (but it is still illegal to call `upc_all_fcntl` with any other argument when an asynchronous operation is outstanding).

7.3.3 Reading Data

Common Constraints

1. No function in this section (7.3.3) may be called while an asynchronous operation is pending on the file handle.

7.3.3.1 The `upc_all_fread_local` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_local(upc_file_t *fd,
                            void *buffer,
                            size_t size,
                            size_t nmemb,
                            upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_local` reads data from a file into a local buffer on each thread.

2. This function can be called only if the current file pointer type is an individual file pointer, and the file handle is open for reading.

3. `buffer` is a pointer to an array into which data will be read, and each thread may pass a different value for `buffer`.

4. Each thread reads `(size*nmemb)` bytes of data from the file at the position indicated by its individual file pointer into the buffer. Each thread may pass a different value for `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O.

5. On success, the function returns the number of bytes read into the local buffer of the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns –1 and sets `errno` appropriately.

7.3.3.2 The `upc_all_fread_shared` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_shared(upc_file_t *fd,
                             shared void *buffer,
                             size_t blocksize,
                             size_t size,
                             size_t nmemb,
                             upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_shared` reads data from a file into a shared buffer in memory.

2. The function can be called when the current file pointer type is either a common file pointer or an individual file pointer. The file handle must be open for reading.

3. `buffer` is a pointer to an array into which data will be read. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of 0 indicates an indefinite blocking factor.

4. In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. Each thread reads `(size*nmemb)` bytes of data from the file at the position indicated by its individual file pointer into its buffer. Each thread may pass a different value for `blocksize`, `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes read

by the calling thread, and the individual file pointer of the thread is incremented by that amount.

5. In the case of a common file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter, and the same value for `blocksize`, `size` and `nmemb`. The effect is that `(size*nmemb)` bytes of data are read from the file at the position indicated by the common file pointer into the buffer. If `size` or `nmemb` is zero, the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes read by all threads, and the common file pointer is incremented by that amount.

6. If reading with individual file pointers results in overlapping reads into the shared buffer, the result is determined by whether the file was opened with the `UPC_STRONG_CA` flag or not (see Section 7.3.2.1).

7. The function returns –1 on error and sets `errno` appropriately.

7.3.4 Writing Data

Common Constraints

1. No function in this section (7.3.4) may be called while an asynchronous operation is pending on the file handle.

7.3.4.1 The `upc_all_fwrite_local` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_local(upc_file_t *fd,
                             void *buffer,
                             size_t size,
                             size_t nmemb,
                             upc_flag_t sync_mode)
```

Description

1. upc_all_fwrite_local writes data from a local buffer on each thread into a file.

2. This function can be called only if the current file pointer type is an individual file pointer, and the file handle is open for writing.

3. buffer is a pointer to an array from which data will be written, and each thread may pass a different value for buffer.

4. Each thread writes (size*nmemb) bytes of data from the buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for size and nmemb. If size or nmemb is zero, the buffer argument is ignored and that thread performs no I/O.

5. If any of the writes result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for fd (see Section 7.3.2.1).

6. On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns –1 and sets errno appropriately.

7.3.4.2 The upc_all_fwrite_shared function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_shared(upc_file_t *fd,
                              shared void *buffer,
                              size_t blocksize,
                              size_t size,
                              size_t nmemb,
                              upc_flag_t sync_mode)
```

Description

1. upc_all_fwrite_shared writes data from a shared buffer in memory to a file.

2. The function can be called if the current file pointer type is either a common file pointer or an individual file pointer. The file handle must be open for writing.

3. `buffer` is a pointer to an array from which data will be written. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of 0 indicates an indefinite blocking factor.

4. In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. Each thread writes `(size*nmemb)` bytes of data from its buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for `blocksize`, `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount.

5. In the case of a common file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter, and the same value for `blocksize`, `size` and `nmemb`. The effect is that `(size*nmemb)` bytes of data are written from the buffer to the file at the position indicated by the common file pointer. If `size` or `nmemb` is zero, the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes written by all threads, and the common file pointer is incremented by that amount.

6. If writing with individual file pointers results in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).

7. The function returns –1 on error and sets `errno` appropriately.

7.3.5 List I/O

Common Constraints

1. List I/O functions take a list of addresses/offsets and corresponding lengths in memory and file to read from or write to.

2. List I/O functions can be called regardless of whether the current file pointer type is individual or common.

3. File pointers are not updated as a result of a list I/O read/write operation.

4. The `memvec` argument passed to any list I/O *read* function by a single thread must not specify overlapping regions in memory.

5. The base addresses passed to `memvec` can be in any order.

6. The `filevec` argument passed to any list I/O *write* function by a single thread must not specify overlapping regions in the file.

7. The offsets passed in `filevec` must be in monotonically non-decreasing order.

8. No function in this section (7.3.5) may be called while an asynchronous operation is pending on the file handle.

9. No function in this section (7.3.5) implies the presence of barriers at entry or exit. However, the programmer is advised to use a barrier after calling `upc_all_fread_list_shared` to ensure that the entire shared buffer has been filled up, and similarly, use a barrier before calling `upc_all_fwrite_list_shared` to ensure that the entire shared buffer is up-to-date before being written to the file.

10. For all the list I/O functions, each thread passes an independent set of memory and file vectors. Passing the same vectors on two or more threads specifies redundant work. The file pointer is a single-valued argument, all other arguments to the list I/O functions are NOT single-valued.

7.3.5.1 The `upc_all_fread_list_local` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_list_local(upc_file_t *fd,
                                             size_t memvec_entries,
                                             upc_local_memvec_t const *memvec,
                                             size_t filevec_entries,
                                             upc_filevec_t const *filevec,
                                             upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_list_local` reads data from a file into local buffers in memory. The file handle must be open for reading.

2. `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3. The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

4. On success, the function returns the number of bytes read by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.3.5.2 The `upc_all_fread_list_shared` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_list_shared(upc_file_t *fd,
                                  size_t memvec_entries,
                                  upc_shared_memvec_t const *memvec,
                                  size_t filevec_entries,
                                  upc_filevec_t const *filevec,
                                  upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_list_shared` reads data from a file into various locations of a shared buffer in memory. The file handle must be open for reading.

2. `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3. The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

4. If any of the reads from different threads result in overlapping regions in memory, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).

5. On success, the function returns the number of bytes read by the calling thread. On error, it returns –1 and sets `errno` appropriately.

   Note: With the above definition, there is no way to do with explicit offsets the equivalent of `upc_all_fread_shared` *using a common file pointer, namely, where all threads specify the same access (same parameters), the data gets read collectively into the shared buffer, and the function returns the total amount of data read by all threads.*

7.3.5.3 The `upc_all_fwrite_list_local` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_list_local(upc_file_t *fd,
                                  size_t memvec_entries,
                                  upc_local_memvec_t const *memvec,
                                  size_t filevec_entries,
                                            upc_filevec_t const *filevec,
                                            upc_flag_t sync_mode)
```

Description

1. `upc_all_fwrite_list_local` writes data from local buffers in memory to a file. The file handle must be open for writing.

2. `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3. The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

4. If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).

5. On success, the function returns the number of bytes written by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.3.5.4 The `upc_all_fwrite_list_shared` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_list_shared(upc_file_t *fd,
                                   size_t memvec_entries,
                                   upc_shared_memvec_t const *memvec,
                                   size_t filevec_entries,
                                   upc_filevec_t const *filevec,
                                   upc_flag_t sync_mode)
```

Description

1. `upc_all_fwrite_list_shared` writes data from various locations of a shared buffer in memory to a file. The file handle must be open for writing.

2. `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3. The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

4. If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).

5. On success, the function returns the number of bytes written by the calling thread. On error, it returns –1 and sets `errno` appropriately.

   Note: With the above definition, there is no way to do with explicit offsets the equivalent of `upc_all_fwrite_shared` *using a common file pointer, namely, where all threads specify the same access (same parameters), the data gets written collectively from a shared buffer, and the function returns the total amount of data written by all threads.*

7.3.6 Asynchronous I/O

Common Constraints

1. Only one asynchronous I/O operation can be outstanding on a UPC-IO file handle at any time. If an application attempts to initiate a second asynchronous I/O operation while one is still outstanding on the same file handle the behavior is undefined – however, high-quality implementations will issue a fatal error.

2. For asynchronous read operations, the contents of the destination memory are undefined until after a successful `upc_all_fwait_async` or `upc_all_ftest_async` on the file handle. For asynchronous write operations, the source memory may not be safely modified until after a successful `upc_all_fwait_async` or `upc_all_ftest_async` on the file handle.

3. An implementation is free to block for completion of an operation in the asynchronous initiation call or in the `upc_all_ftest_async` call (or both). High-quality implementations are recommended to minimize the amount of time spent within the asynchronous initiation or `upc_all_ftest_async` call.

4. In the case of list I/O functions, the user may modify or free the lists after the asynchronous I/O operation has been initiated.

7.3.6.1 The `upc_all_fread_local_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

void upc_all_fread_local_async(upc_file_t *fd,
                               void *buffer,
                               size_t size,
                               size_t nmemb,
                               upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_local_async` initiates an asynchronous read from a file into a local buffer on each thread.

2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_local`.

3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.2 The `upc_all_fread_shared_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

void upc_all_fread_shared_async(upc_file_t *fd,
                                shared void *buffer,
                                size_t blocksize,
                                size_t size,
                                size_t nmemb,
                                upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_shared_async` initiates an asynchronous read from a file into a shared buffer.
2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_shared`.
3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.3 The `upc_all_fwrite_local_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_local_async(upc_file_t *fd,
                                void *buffer,
                                size_t size,
                                size_t nmemb,
                                upc_flag_t sync_mode)
```

Description

1. `upc_all_fwrite_local_async` initiates an asynchronous write from a local buffer on each thread to a file.

2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_local`.

3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.4 The `upc_all_fwrite_shared_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_shared_async(upc_file_t *fd,
                                 shared void *buffer,
                                 size_t blocksize,
                                 size_t size,
                                 size_t nmemb,
                                 upc_flag_t sync_mode)
```

Description

1. `upc_all_fwrite_shared_async` initiates an asynchronous write from a shared buffer to a file.

2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_shared`.

3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.5 The `upc_all_fread_list_local_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>
```

```
void upc_all_fread_list_local_async(upc_file_t *fd,
                                    size_t memvec_entries,
                                    upc_local_memvec_t const *memvec,
                                    size_t filevec_entries,
                                    upc_filevec_t const *filevec,
                                    upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_list_local_async` initiates an asynchronous read of data from a file into local buffers in memory.

2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_list_local`.

3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.6 The `upc_all_fread_list_shared_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

void upc_all_fread_list_shared_async(upc_file_t *fd,
                                     size_t memvec_entries,
                                     upc_shared_memvec_t const *memvec,
                                     size_t filevec_entries,
                                     upc_filevec_t const *filevec,
                                     upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_list_shared_async` initiates an asynchronous read of data from a file into various locations of a shared buffer in memory.

2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_list_shared`.

3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.7 The `upc_all_fwrite_list_local_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_list_local_async(upc_file_t *fd,
                                     size_t memvec_entries,
                                     upc_local_memvec_t const *memvec,
                                     size_t filevec_entries,
                                     upc_filevec_t const *filevec,
                                     upc_flag_t sync_mode)
```

Description

1. `upc_all_fwrite_list_local_async` initiates an asynchronous write of data from local buffers in memory to a file.
2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_list_local`.
3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.8 The `upc_all_fwrite_list_shared_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_list_shared_async(upc_file_t *fd,
                                      size_t memvec_entries,
                                      upc_shared_memvec_t const *memvec,
                                      size_t filevec_entries,
                                      upc_filevec_t const *filevec,
                                      upc_flag_t sync_mode)
```

Description

1. `upc_all_fwrite_list_shared_async` initiates an asynchronous write of data from various locations of a shared buffer in memory to a file.
2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_list_shared`.
3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.3.6.9 The `upc_all_fwait_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwait_async(upc_file_t *fd)
```

Description

1. `upc_all_fwait_async` completes the previously issued asynchronous I/O operation on the file handle `fd`, blocking if necessary.
2. It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.
3. On success, the function returns the number of bytes read or written by the asynchronous I/O operation as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns –1 and sets `errno` appropriately, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

7.3.6.10 The `upc_all_ftest_async` function

Synopsis

1.
```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_ftest_async(upc_file_t *fd,
                            int *flag)
```

Description

1. `upc_all_ftest_async` tests whether the outstanding asynchronous I/O operation associated with `fd` has completed.

2. If the operation has completed, the function sets `flag=1` and the asynchronous operation becomes no longer outstanding;[8] otherwise it sets `flag=0`. The same value of `flag` is set on all threads.

3. If the operation was completed, the function returns the number of bytes that were read or written as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns –1 and sets `errno` appropriately, and sets the `flag=1`, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

4. It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.

[8]This implies it is illegal to call `upc_all_fwait_async` or `upc_all_ftest_async` immediately after a successful `upc_all_ftest_async` on that file handle.

References

[1] Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specifications V1.1.1 (http://upc.gwu.edu), October 2003.

[2] ISO Programming Languages-C. ISO/IEC 9899:1999(E), May 2000.

[3] MPI-2: Extensions to the Message-Passing Interface, Message Passing Interface Forum, July 18, 1997.

[4] Elizabeth Wiebel, David Greenberg and Steven Seidel, UPC Collective Operations Specification V1.0 (http://www.gwu.edu/ upc/docs/UPC_Coll_Spec_V1.0.pdf), December 2003.

APPENDIX A : Future Library Directions

This section describes features that will be discussed in future releases of the UPC-IO Specifications (but have been explicitly tabled for the current release).

1. Add support for allowing multiple outstanding asynchronous operations on the same file handle, to allow more aggressive computational overlap with file operations. This can be done by introducing a handle data type like `upc_all_handle_t` to represent the explicitly non-blocking collective I/O operation in flight, and have this value returned by each async init function and consumed by explicit-handle synchronization functions (e.g., `upc_all_handle_test()`/`upc_all_handle_wait()`). All of the asynchronous initiation functions currently return `void`, so we could add this new handle return type without breaking any existing code. In order to be backwards compatible with the current interface, the default behavior should remain at the current behavior (allow only a single outstanding async operation, synchronized using `upc_all_ftest_async()`/`upc_all_fwait_async()`) and we can use a new `upc_all_fcntl` setting to enable multi-operation handle-based async I/O as described here. We may even want to use the same handle data type and sync functions for explicitly non-blocking UPC collectives.

   An alternate approach to allowing multiple outstanding asynchronous operations on the same file handle would be an implicit-handle approach, where we keep the current interface and simply lift the restriction that only one asynchronous operation can be in-flight per handle. This approach offers the client less flexibility in synchronization (because the only choice is to sync all outstanding operations, rather than a particular subset), but it may be an acceptable compromise. However, we'd have to think about how errors would be reported by the synchronization functions which complete more than one operation.

   Regardless of the design chosen, an important semantic issue that must be resolved when more than one async call can be in-flight is specifying exactly when and how the file pointer is updated by an async operation, especially in the presence of errors or reading to the EOF. The semantics of the second and subsequent async I/O operations are not well-defined unless we specify how the file pointer is affected by the async

I/O operations already in-flight. One possibility for side-stepping this issue is to only allow multiple outstanding async I/O operations of the list I/O variety, which are completely independent of the problematic file pointer.

2. `upc_all_fcntl` currently provides the means to change most of the interesting `upc_all_fopen` flags in effect for the given file handle. The only `upc_all_fopen` flag which persists as an attribute of the file handle and currently cannot be changed after open is the read-only/write-only/read-write status of the file handle. Do we want to support changing this via a `upc_all_fcntl`? Is it a useful capability? What are the implementation issues? (at worst it seems this could always be implemented with a close and reopen). Note that C99 provides this capability via `freopen()`, and UPC-IO currently has no equivalent – however, the C99 semantics are too weak to be portably reliable: "If filename is a null pointer, the `freopen()` function attempts to change the mode of the stream to that specified by mode, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances." It's also unclear from the spec what the required behavior of the file pointer is on such an `freopen()`. If we decide to provide this capability, we should be less wishy-washy about the semantics to ensure it's portably usable.